Overview

What is recursion?
- When one function calls ITSELF directly or indirectly.

Why learn recursion?
- New mode of thinking.
- Powerful programming tool.
- Many computations are naturally self-referential.
  - a Unix directory contains files and other directories
  - Euclid’s gcd algorithm
  - linked lists and trees
  - GNU = GNU’s Not Unix

Drawing Hands
M. C. Escher, 1948

Overview

How does recursion work?

How does a function call work?
- A function lives in a local environment:
  - values of local variables
  - which statement the computer is currently executing
- When f() calls g(), the system
  - saves local environment of f
  - sets value of parameters in g
  - jumps to first instruction of g, and executes that function
  - returns from g, passing return value to f
  - restores local environment of f
  - resumes execution in f just after the function call to g

Implementing Functions

How does the compiler implement functions?

Return from functions in last-in first-out (LIFO) order.
- FUNCTION CALL: push local environment onto stack.
- RETURN: pop from stack and restore local environment.
A Simple Example

Goal: function to compute \( \text{sum}(n) = 0 + 1 + 2 + \ldots + n-1 + n \)

- Simple ITERATIVE solution.

\begin{verbatim}
iterative sum 1
int sum(int n) {
    int i, s = 0;
    for (i = 0; i <= n; i++)
        s += i;
    return s;
}
\end{verbatim}

\begin{verbatim}
iterative sum 2
int sum(int n) {
    int s = n;
    while (n > 0) {
        n--;
        s += n;
    }
    return s;
}
\end{verbatim}

Note that changing the variable \( n \) in \text{sum} does not change the value in the calling function.

A Simple Example

Goal: function to compute \( \text{sum}(n) = 0 + 1 + 2 + \ldots + n-1 + n \).

- Simple ITERATIVE solution.
- Can also express using SELF-REFERENCE.

This is just a stupid example to illustrate recursion.
- Don’t even need iteration, let alone recursion.
- \( 0 + 1 + 2 + \ldots + n = n(n+1) / 2 \)

\begin{verbatim}
better sum
int sum(int n) {
    return (n * (n+1)) / 2;
}
\end{verbatim}

A Bad Recursive Function

BASE CASE is special input for which the answer is trivial.
- Won’t "bottom-out" of recursion without a base case.
- Analog of infinite loops with for and while loops.

\begin{verbatim}
mystery1(n)
void mystery1(int n) {
    printf("%d\n", n);
    if (n % 2 == 0)
        mystery1(n/2);
    else
        mystery1(3*n + 1);
}
\end{verbatim}

No base case
A Bad Recursive Function

BASE CASE is special input for which the answer is trivial.

REDUCTION STEP makes input converge to base case.
  - Unknown whether program terminates for all positive integers n.
  - Stay tuned for Halting Problem in computability lecture.

```c
void mystery2(int n)
{
    printf("%d\n", n);
    if (n <= 1)
        return;
    else if (n % 2 == 0)
        mystery2(n/2);
    else
        mystery2(3*n + 1);
}
```

Greatest Common Divisor

Find largest integer d that evenly divides into m and n.

$$\gcd(m, n) = \begin{cases} 
    m & \text{if } n = 0 \\
    \gcd(n, m \% n) & \text{otherwise}
\end{cases}$$

<table>
<thead>
<tr>
<th>m</th>
<th>n</th>
<th>m % n</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcd</td>
<td>gcd</td>
<td>gcd</td>
</tr>
</tbody>
</table>

- base case
- reduction step
- converges to base case

m = 8x
n = 3x
\(\gcd(m, n) = x\)

Euclid (300 BCE)

gcd(1440, 408) = gcd(408, 216) = gcd(216, 192) = gcd(192, 24) = gcd(24, 0) = 24.

1440 = 2^5 \times 3^2 \times 5^1
408 = 2^3 \times 3^1 \times 17^1
Number Conversion

To print binary representation of integer N:
- Stop if N = 0.
- Write ‘1’ if N is odd; ‘0’ if N is even.
- Move pencil one position to left.
- Print binary representation of N / 2.

(integer division)

Check: \(43 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0\)

= 32 + 0 + 8 + 0 + 2 + 1

Easiest way to compute by hand.
- Corresponds directly with a recursive program.

Recursive Number Conversion

Computer naturally prints from left to right.
- So we need to first convert N / 2.
- Then write ‘0’ or ‘1’.

Proof of correctness:
\(N = 2 \times (N / 2) + (N \% 2)\)

Convert to any base \(b \leq 10\).
- Exercise: extend to handle hexadecimal (base 16).

Fibonacci Numbers

Infinite series: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

A natural for recursion.

\[
F_n = \begin{cases} 
0 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
F_{n-1} + F_{n-2} & \text{otherwise}
\end{cases}
\]

Fibonacci Rabbits:
L. P. Fibonacci (1170 - 1250)
**Possible Pitfalls With Recursion**

**Is recursion fast?**

-Fibonacci numbers: 0, 1, 2, 3, 5, 8, 13, 21, 34, ...

-It takes a really long time to compute F(40).

---

**bad Fibonacci function**

```c
int F(int n) {
    if (n == 0 || n == 1)
        return n;
    else
        return F(n-1) + F(n-2);
}
```

---

**It takes a really long time to compute F(40).**

---

**bad Fibonacci function**

```c
int F(int n) {
    if (n == 0 || n == 1)
        return n;
    else
        return F(n-1) + F(n-2);
}
```

---

**Fibonacci using dynamic programming**

```c
int F(int n) {
    if (knownF[n] != 0)
        return knownF[n];
    else if (n == 0 || n == 1)
        return n;
    else
        knownF[n] = F(n-1) + F(n-2);
    return knownF[n];
}
```

---

**Recursion vs. Iteration**

**Fact 1.** Any recursive function can be written with iteration.
- Compiler implements recursion with stack.
- Can avoid recursion by explicitly maintaining a stack.

**Fact 2.** Any iterative function can be written with recursion.

Should I use iteration or recursion?
- Ease and clarity of implementation.
- Time/space efficiency.
**Towers of Hanoi**

Move all the discs from the leftmost peg to the rightmost one.
- Only one disc may be moved at a time.
- A disc can be placed either on empty peg or on top of a larger disc.

**Towers of Hanoi demo**

Edouard Lucas (1883)

---

**Towers of Hanoi: Recursive Solution**

```c
#include <stdio.h>

void hanoi(int n, char from, char to) {
    char temp;
    if (n == 0) return;
    temp = 'A' + 'B' + 'C' - from - to;
    hanoi(n-1, from, temp);
    printf("Move disc %d from %c to %c.\n", n, from, to);
    hanoi(n-1, temp, to);
}

int main(void) {
    hanoi(4, 'A', 'C');
    return 0;
}
```

**Unix**

```
% gcc hanoi.c
% a.out
Move disc 1 from A to B.
Move disc 2 from A to C.
Move disc 1 from B to C.
Move disc 3 from A to B.
Move disc 1 from C to A.
Move disc 2 from C to B.
Move disc 1 from A to B.
Move disc 4 from A to C.
Move disc 1 from B to C.
Move disc 2 from B to A.
Move disc 1 from C to A.
Move disc 3 from B to C.
Move disc 1 from A to B.
Move disc 2 from A to C.
Move disc 1 from B to C.
```
Towers of Hanoi

Is world going to end (according to legend)?
- Monks have to solve problem with \( N = 40 \) discs.
- Computer algorithm should help.

Better understanding of recursive algorithm supplies non-recursive solution!
- Alternate between two moves:
  - Move smallest disc 1 peg to right (left) if \( N \) is even (odd).
  - Make only legal move not involving smallest disc.

See Sedgewick 5.2.

Summary

How does recursion work?
- Just like any other function call.

How does a function call work?
- Save away local environment using a stack.

Trace the executing of a recursive program.
- Use pictures.

Write simple recursive programs.
- Base case.
- Reduction step.

Write recursive electronic music?
- *Towers of Hanoi* by W. A. Schloss.

Mathematical Induction

Mathematical induction.
- Powerful and general proof technique in discrete mathematics.
- To prove a theorem true for all integers \( N \geq 0 \):
  - Base case: Prove it to be true for \( N = 0 \).
  - Induction step: Assuming it is true for all \( k < N \), prove it is true for \( N \).

Theorem: \( 0 + 1 + 2 + 3 + \ldots + N = \frac{N(N+1)}{2} \) for all \( N \geq 0 \).
Proof: (by mathematical induction)
- Base case (\( N = 0 \)).
  - \( 0 = 0(0+1)/2 \).
- Induction step.
  - Assume \( 0 + 1 + 2 + \ldots + k = \frac{k(k+1)}{2} \) for all \( 0 \leq k < N \).
  - \( 0 + 1 + 2 + \ldots + N-1 + N = \frac{(0 + 1 + 2 + \ldots + N-1) + N}{2} \)
  - \( = \frac{((N-1) N / 2) + N}{2} \)
  - \( = \frac{N(N+1)}{2} \)
Mathematical Induction and Recursion

Mathematical induction and programming.
- Prove correctness of recursive functions.

```
0 + 1 + 2 + ... + n
int sum(int n) {
    if (n == 0) return 0;
    else return n + sum(n-1);
}
```
- Find alternate non-recursive computation.

```
0 + 1 + 2 + ... + n
int sum(int n) {
    return (n * (n+1)) / 2;
}
```

Exponentiation

Goal: function to compute $x^N$, for positive integers $x$, $N$.
- Simple ITERATIVE solution.

```
iterative power function
int power(int x, int N) {
    int i, prod = 1;
    for (i = 0; i < N; i++) {
        prod *= x;
        return prod;
    }
}
```
- Can also express using SELF-REFERENCE.

```
recursive power function
int power(int x, int N) {
    if (0 == N) return 1;
    else return x * power(x, N-1);
}
```
- Both require $N$ multiplications, but can do with $N/2 + 1$ if $N$ is even.

```
x^N = \begin{cases} 
    1 & \text{if } N = 0 \\
    x \cdot x^{N-1} & \text{otherwise} 
\end{cases}
```

Exponentiation

Goal: function to compute $x^N$, for positive integers $x$, $N$.
- Simple ITERATIVE solution.
- Can also express using SELF-REFERENCE.
- Both require $N$ multiplications, but can do with $N/2 + 1$ if $N$ is even.
Exponentiation

Goal: function to compute $x^N$, for positive integers $x, N$.

- Simple ITERATIVE solution.
- Can also express using SELF-REFERENCE.
- Both require $N$ multiplications, but can do with $N/2 + 1$ if $N$ is even.
- Only $2 \log_2 N$ multiplications needed with divide-and-conquer!

```
int power(int x, int N) {
    int t;
    if (N == 0)
        return 1;
    t = power(x, N/2);
    if (N % 2 == 0)
        return t * t;
    else
        return x * t * t;
}
```

Fast exponentiation crucial for cryptography.
- Similar technique for multiplying large integers.
- Stay tuned: RSA crypto assignment.